Autonomous Underwater Vehicles: Hybrid Control of Mission and Motion

D.B. MARCO, A.J. HEALEY AND R.B. McGHEE

Autonomous Underwater Vehicle Laboratory, Naval Postgraduate School, 700, Dyer Road, Monterey, CA 93943

healey@me.nps.navy.mil

Abstract. This paper provides an experimental implementation and verification of a hybrid (mixed discrete state/ continuous state) controller for semi-autonomous and autonomous underwater vehicles in which the missions imply multiple task robot behavior. An overview of some of the missions being considered for this rapidly developing technology is mentioned including environmental monitoring, underwater inspection, geological survey as well as military missions in mine countermeasures.

The functionalities required of such vehicles and their relation to 'intelligent control' technology is discussed. In particular, the use of Prolog as a computer language for the specification of the discrete event system (DES) aspects of the mission control is proposed. The connections between a Prolog specification and the more common Petri Net graphical representation of a DES are made. Links are made between activation commands, transitioning signals, and the continuous state dynamic control system (DCS) responsible for vehicle stabilization.

Details are given of the NPS Phoenix vehicle implementation at the present time, together with experimental validation of the concepts outlined using a simplified example mission. The paper ends with a listing of questions and concerns for the evaluation of software controllers. A list of references is given for readers interested in this subject.

Keywords: autonomous, hybrid control, Prolog, Petri nets, underwater vehicles, reactive, sonar, mission coordination

1. Introduction

Spawned from the availability of small embedded processors and the increasing capabilities of underwater communications, small untethered vehicles are expected to play a role in expanding our ability to survey ocean areas. Missions are expected to include environmental monitoring, underwater inspection and monitoring, geological survey as well as military missions in surveillance and mine countermeasures (Yuh, 1994; Moore, 1994; Curtin et al., 1993).

The key to the technology is to provide a vehicle free from the constraints of a physical tether as is currently used by offshore operators and ocean scientists with Remotely Operated Vehicles (ROV's) (Newman and Stakes, 1994). It means that sufficient power and intelligence must be provided onboard while communications with the user will be at a higher level

than is currently the practice with ROVs today. All this is to be achieved using low cost components and maintaining as high a reliability as possible. The relationship of a semi-autonomous/autonomous vehicle to the ROV is an evolutionary one-made possible by advances in acoustic communications and high energy power sources. Coupling a free swimming capability with moderate endurance and the ability to reliably communicate between the vehicle and the human user using acoustics (even at a low rate), enables the command of robot actions from remote locations, and the observation of its behavior and sensory data streams without physical restriction.

The critical features that involve control technology include, underwater navigation, high level command and control through acoustic communications, precision motion control of the vehicle, and coordinated control of vehicle/manipulator motion. Other

	Report Documentation Page	Form Approved OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **1996**	2. REPORT TYPE	3. DATES COVERED **-**

4. TITLE AND SUBTITLE **Autonomous Underwater Vehicles: Hybrid Control of Mission and Motion**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Postgraduate School,Center for AUV Research,Monterey,CA,93943-5000**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES **20**	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**			

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

important features are enhanced reliability through autonomous adaptive resource reconfiguration and in-built failure diagnostics with error recovery procedures at all levels (Healey et al., 1991).

The development of intelligent control technology for underwater robots lies, for the moment, at the intersection of Discrete Event Systems (DES) and Dynamic Control of Continuous Systems (DCS) where system theory is well developed for each alone but not both acting together. It is not well understood how to formally evaluate the performance of these combined systems that are now being referred to as 'Hybrid' control systems (Antsaklis and Passino, 1993). Saridis (1989) introduced the concept of 'entropy' for a multidimensional performance index that could possibly be optimized for hybrid systems. Computer Aided Design of these systems has been proposed (Simon et al., 1993) using a rigid robot manipulator as an example, overcoming the lack of formal methods by using ORCAD, a CAD package and the synchronous language 'Esterel' developed especially for handling DES as automata.

Software architectures for underwater vehicles—a distinctly different problem from robotic manipulators—involve vehicle stabilization issues, and have been described and discussed in previous literature (Hall and Adams, 1992; Albus, 1988; Sousa et al., 1994), but without any experimental validation. Few detailed results have been quantified for the Odyssey class of vehicle (Smith and Dunn, 1994; Bellingham et al., 1994) although the Odyssey has performed under ice and demonstrated homing behaviors into a capture net.

Some Hybrid systems are predominantly DES and can be designed using state tables and finite state machines, or recently, Petri net methodologies (Cassandras,1993). Others are predominantly continuous DCS with only a small component of discrete state logic for which stability theory and well established optimal control techniques are well suited (Friedland, 1986). 'Hybrid', in the context of this paper, deals with the underwater robot control problem which is a true mix of DES and DCS for which new design techniques and evaluation methods are needed. In order to separate the functionality of the system we note that the control of the sequencing of a mission is a discrete event system (DES) problem with the state transitions driven by conditions arising from the completion of robot tasks or by sensor based events, while the stabilization and control of vehicle motion to mission derived trajectories and or set points, is a traditional problem in dynamic control.

The NASREM architecture (Albus and Quintero, 1990) is at one end of the spectrum of Hybrid controllers and relies on a hierarchical system of planning. At the other end is the layered control with subsumption (Brooks, 1986) modified with discrete state coordination as in Bellingham and Consi (1991). The state transitions arise from completion of robot tasks while the specifications of a mission phase generates plans for vehicle motions in terms of either set points and control mode activations. It is the later that forms the basis for linking the mission control (DES) at the top (Strategic Level) to the vehicle control (DCS) at the bottom, (Execution Level) and is embodied in a middle (Tactical Level) set of control software functions.

We have thus defined a tri-level software control architecture (—the Rational Behavior Model—(Byrnes et al., 1992, 1993)) comprising Strategic, Tactical, and Execution levels. The three levels separate the software into easily modularized functions encompassing everything from logically intense discrete state transitioning through the interfacing of *asynchronous* data updates with the real time *synchronized* controller functions that stabilize the vehicle motion to set points or trajectory commands.

The distinguishing features that identify each level are

1. *Strategic Level*:   This level in the architecture uses a rule based language and is entirely boolean—dealing only with the management of the discrete state transitioning required to perform the mission control. No numerical computations are done at this level and no memory is required except for the state of the mission. In principle, it determines what needs to be done next.

2. *Execution Level*:   This level contains all the code functions that are required to stabilize the motion control of the vehicle to a set of commands that could be modes to be activated and servo set points where that servo control functions can be complex even including command overrides for reflexive behavior and adaptive control features. Many robot controllers stop at this level.

3. *Tactical Level*:   This level is a set of functions that are compiled as primitive predicates in the Strategic Level Rules which open and close lines of communications between the Strategic Level and the Execution Level functions. They include the functions that gather data from the servo level and perform the necessary computations to determine if the

robot tasks are completed, perform the navigational planning functions, the sonar computations and evaluates and sends appropriate setpoints and servo mode activation flags to the Execution Level. In this level, the computations are numerical but asynchronous with respect to time. The distinguishing feature between the Tactical and Execution Level software is that of the need for hard real time completion in the Exection Level and asynchronous completion in the Tactical Level.

In our controller architecture, the Strategic level uses 'Prolog' as a rule based mission control specification language. Other DES control system design techniques and implementation methods could be used, although, it is the experience of the authors that none is more convenient than using this existing language. Prolog has the advantage of being an executable specification language which can run in real time as we demonstrate herein. The DES represented by the mission specification could have been be translated in to 'C' code or into 'Ada' or other languages such as Esterel (Simon et al., 1993) and 'Coral' (Silva et al., 1994). However, in our use of Prolog, the Prolog inference engine cycles through the predicate rules and in doing so, manages the state transition aspects of mission control so the need for logical verification of the control specification disappears. It transitions the states in real time, and generally develops the commands (activations) that drive the vehicle through its mission. Error recovery procedures from failures in the mission tasks or the vehicle subsystems are handled as transitions to 'error' states that ultimately provide commands to the servo level control for appropriate recovery action.

The Tactical Level, currently written in C is set of functions that are linked at compile time with the Prolog predicates and are designed to either return TRUE / FALSE in response to queries—these are distinguished by the prefix 'Ask' in the Prolog rules—or to activate commands, distinguished by the prefix 'Exec'. These Tactical Level functions are also interfaced to the real time Execution Level controller using asynchronous communications and script type messages passed through an ethernet socket with TCP/IP protocol.

The Execution Level controller is designed to command the vehicle subsystems appropriately for the mode flags and set points sent on the socket and to activate robot behaviors that correspond to those commanded. Communication from the Tactical Level to the Execution Level takes place through a single socket. By the design of this hierarchical control system, the

Tactical Level runs asynchronously and retains the mission data file and the mission log file in global memory. It sends the command scripts to the Execution Level and requests data for the evaluation of state transitions. The architecture is a hybrid between the true hierarchical control of NASREM (Albus and Quintero, 1990) and purely reactive of subsumption (Brooks, 1986) schemes. In this way, control of mission is retained, while reacting to unanticipated events is also enabled.

While earlier results at NPS were obtained by workstation simulations (Byrnes, 1993), we have now implemented the tri-level hybrid control in the Phoenix vehicle (Healey et al., 1994) with real time results in complex missions.

The major contribution of this paper will be to describe some of many results of real time hybrid control experiments with the NPS Phoenix vehicle, obtained recently, in which *all three levels of the control software are active in real time*. We will discuss our results with respect to a simple example mission using the hovering mode capabilities of the vehicle including coordination of the activation of a high frequency profiling sonar as part of the mission plan.

2. System Overview

The Vehicle

The NPS Phoenix, shown in Fig. 1, has been recently outfitted with the tri-level controller and a sonar suite consisting of a Datasonics PSA 900 sonar at 200 kHz. to derive altitude above bottom signals, and two Tritech high frequency sonars that are mechanically scannable through 360 degrees. The ST1000 is a 1024 kHz, 1.5 degree pencil beam profiling sonar that is best suited for measurment of the time of flight for the first stong return. The ST725 is a 725 kHz, 24 degree high by 2.5 degree wide beam scanning sonar which returns intensity as a function of range in bins for any given heading. Both sonars may be mechanically scanned either through 360 degrees, or through a reduced sector, to concentrate on specific obstacles/targets around the vehicle.

These devices may be addressed for control purposes through a serial port where ASCII characters are used to command primitive control functions such as '*send one ping and analyze the return structure*' (either range to first return or intensity in a series of range bins), or '*turn by one step*'. By issuing a sequence of such commands, the sonar head may be made to *self center*,
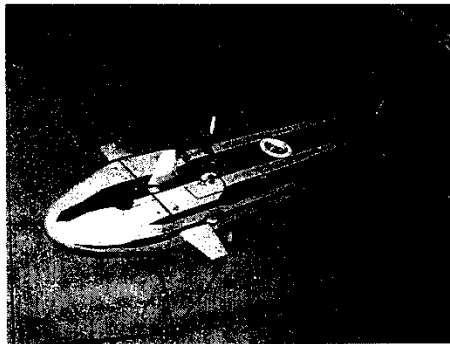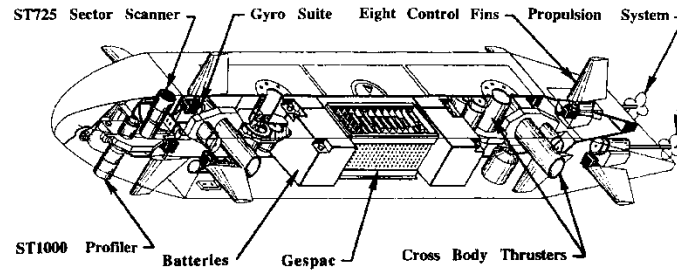
Figure 1. The NPS Phoenix vehicle.

continuously rotate while pinging and return the data stream, or to *sweep over a defined sector and return the data stream.*

Additionally, the vehicle has now been equipped with cross body thrusters. Two vertical thrusters are for heave and pitch control, and two transverse thrusters are for heading and lateral movement control. These complement the two propulsion motors at the stern and eight fin surfaces for flight control. The vehicle has a dry interior and a wet nose, a length of 2.13 meters and a dry weight of 175 kg. Sufficient energy storage (1100 Whr.) is provided by 4 lead acid gel batteries for approximately 3 hours of operational testing.

Vehicle Primitives

In previous work (Healey and Marco, 1992a), waypoint following in a transit phase of a mission was demonstrated in a swimming pool test area where stable behaviors of the vehicle were demonstrated including

(a) Forward Speed Control,
(b) Fin_Steering
(c) Fin_Depth_Control

(d) Waypoint_Following
(e) Bottom_Following, and
(f) Obstacle_Avoidance.

These control functions were implemented with (a)–(c) and (f) running simultaneously, but subsumed by the guidance laws implemented in (d); and, with (c) subsumed by (e). The control laws implemented have been based on PD, and Sliding Mode methods as explained in Healey and Marco (1992b).

Control laws for these functions are readily implemented entirely in the Execution Level with digital control algorithms running at 0.1 sec. update rate. Now, however, new, more complex functions are being enabled using active control of thrusters and sonar. These are,

(g) Submerge_and_Pitch_Control
(h) Heading_Control
(i) Longitudinal_Positional_Control
(j) Speed_Control using command generators
(k) Lateral_Positional_Control
(l) Center_Sonar
(m) Ping Sonar (Mode 0)
(n) Ping and Rotate Sonar Clockwise (Mode +1)

(o) Ping and Rotate Sonar Counter-Clockwise (Mode −1)

(p) Ping and Rotate Sonar Through a Sector (Mode 2)

(q) Initiate_Filter_For_Sonar_Range (Needed For Smoothed Range and Range Rate Estimation)

(r) Reinitialize_Filter

Most of these functions need a given subset of the actuator system to be active under the operation of either an open loop command or a feedback control law. Some of the functions use orthogonal sets of actuators and may be activated without conflict. Some use the same actuators to control different functions and thus control laws may be additive. This means, for example, that vertical thrusters may be used via control laws to control depth as well as pitch, and lateral thrusters to control heading as well as lateral position and side slip speed. In combination with propulsion motors, most functions including Submerge_and_Pitch_Control, Longitudinal_Speed_Control and Longitudinal_Position_Control, as well as Heading_Control, may now be commanded reliably. Heading_Control, Submerge _and_Pitch_Control, and virtually any multiple combination of (a) to (r) above are available to the extent that they do not cause a conflict of actuator control or sensor usage.

Orthogonal Behaviors

Orthogonal behaviors are defined as those control behaviors that use non-interacting subsets of actuators. Even though each may use some common sensory data, orthogonal behavior control functions may be activated simultaneously without conflict. An example would be Heading_Control (using lateral thrusters), Longitudinal_Position_Control (using the propusion motors) and Center_Sonar. Non-orthogonal behaviors use intersecting sets of actuators for control of different error functions and thus control laws can be built up from linear combinations of individual control laws— as used for combined heave and pitch control using vertical thrusters.

Activation of orthogonal behaviors are instituted using a script composed of flags and set points that are a way of communicating between Tactical Level C functions and the real time control loop of the Execution Level. At each pass through the loop, a read is made from the communications socket and an if-else structure is used to determine which set of sensors, actuators and control laws are to be activated during the computation cycle. The same technique is used to flag the

activation of sonars, and filtering actions, and similarly for flags to indicate which data streams are to be written in return.

Reactive Behavior Implementation

Reactive behavior in our controller is handled three ways, similarly to that done in the ORCAD design (Simon et al., 1993).

1. In the Execution Level control loop through *command overrides* following a sensor read, as, for instance, a new obstacle detection requiring an emergency surface or obstacle avoidance (flinch) response,

2. At the Tactical Level, reactive error recovery can be handled by resetting key parameters associated with control signal evaluations. An example is the resetting of a control gain or the inclusion of integral control if a particular error function cannot be stabilized,

3. Reactive behavior is also handled at the Strategic Level for catastrophic faults by transitioning to states that command fatal error recovery procedures such as to surface if, for example, a particular mission phase is not completed after a pre-specified time out and all other techniques have been exhausted.

In the results described here, reactive behavior is built in at the Strategic Level by time and space traps using time out calls. If an allocated time is exceeded, the mission phase fails and the vehicle is commanded to surface. Control overrides are built into the Execution Level to surface the vehicle if battery power is too low or if a leak is detected.

3. The Control Network

The control system, illustrated in its simplest form in Fig. 2, is currently implemented in hardware using three networked processors. All Execution Level software is written in C and runs on a Gespac M68030 processor in a separate card cage inside the boat. Connected in the same card cage is an ethernet card and an array of real time interfacing devices for communications to sensors and actuators indicated in the details of Fig. 3. The Execution Level control code containing a set of functions in a compiled module called 'exec' is downloaded first, opening the communications socket on the Gespac side and waiting for the higher level controller to start.

(o) Ping and Rotate Sonar Counter-Clockwise (Mode −1)

(p) Ping and Rotate Sonar Through a Sector (Mode 2)

(q) Initiate_Filter_For_Sonar_Range (Needed For Smoothed Range and Range Rate Estimation)

(r) Reinitialize_Filter

Most of these functions need a given subset of the actuator system to be active under the operation of either an open loop command or a feedback control law. Some of the functions use orthogonal sets of actuators and may be activated without conflict. Some use the same actuators to control different functions and thus control laws may be additive. This means, for example, that vertical thrusters may be used via control laws to control depth as well as pitch, and lateral thrusters to control heading as well as lateral position and side slip speed. In combination with propulsion motors, most functions including Submerge_and_Pitch_Control, Longitudinal_Speed_Control and Longitudinal_Position_Control, as well as Heading_Control, may now be commanded reliably. Heading_Control, Submerge _and_Pitch_Control, and virtually any multiple combination of (a) to (r) above are available to the extent that they do not cause a conflict of actuator control or sensor usage.

Orthogonal Behaviors

Orthogonal behaviors are defined as those control behaviors that use non-interacting subsets of actuators. Even though each may use some common sensory data, orthogonal behavior control functions may be activated simultaneously without conflict. An example would be Heading_Control (using lateral thrusters), Longitudinal_Position_Control (using the propusion motors) and Center_Sonar. Non-orthogonal behaviors use intersecting sets of actuators for control of different error functions and thus control laws can be built up from linear combinations of individual control laws— as used for combined heave and pitch control using vertical thrusters.

Activation of orthogonal behaviors are instituted using a script composed of flags and set points that are a way of communicating between Tactical Level C functions and the real time control loop of the Execution Level. At each pass through the loop, a read is made from the communications socket and an if-else structure is used to determine which set of sensors, actuators and control laws are to be activated during the computation cycle. The same technique is used to flag the activation of sonars, and filtering actions, and similarly for flags to indicate which data streams are to be written in return.

Reactive Behavior Implementation

Reactive behavior in our controller is handled three ways, similarly to that done in the ORCAD design (Simon et al., 1993).

1. In the Execution Level control loop through *command overrides* following a sensor read, as, for instance, a new obstacle detection requiring an emergency surface or obstacle avoidance (flinch) response,

2. At the Tactical Level, reactive error recovery can be handled by resetting key parameters associated with control signal evaluations. An example is the resetting of a control gain or the inclusion of integral control if a particular error function cannot be stabilized,

3. Reactive behavior is also handled at the Strategic Level for catastrophic faults by transitioning to states that command fatal error recovery procedures such as to surface if, for example, a particular mission phase is not completed after a pre-specified time out and all other techniques have been exhausted.

In the results described here, reactive behavior is built in at the Strategic Level by time and space traps using time out calls. If an allocated time is exceeded, the mission phase fails and the vehicle is commanded to surface. Control overrides are built into the Execution Level to surface the vehicle if battery power is too low or if a leak is detected.

3. The Control Network

The control system, illustrated in its simplest form in Fig. 2, is currently implemented in hardware using three networked processors. All Execution Level software is written in C and runs on a Gespac M68030 processor in a separate card cage inside the boat. Connected in the same card cage is an ethernet card and an array of real time interfacing devices for communications to sensors and actuators indicated in the details of Fig. 3. The Execution Level control code containing a set of functions in a compiled module called 'exec' is downloaded first, opening the communications socket on the Gespac side and waiting for the higher level controller to start.
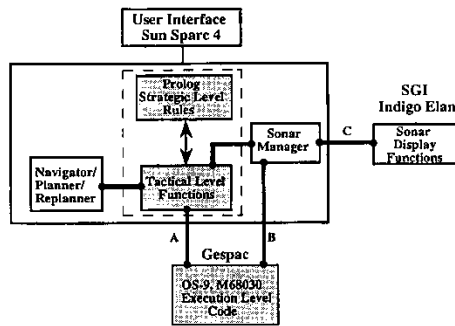
Figure 2. Diagram of the Phoenix networked controller.

Strategic Level

The Strategic Level Prolog rules are compiled and linked together with the supporting Tactical Level C language functions into the single executable process called **'Mission_Control'**, that is run in a Sun Microsystems Sparc 4 laptop computer and linked through an ethernet socket to the Gespac processor (socket 'A' in Fig. 2). Starting **'Mission_Control'** enables communications between both Sparc and Gespac processes.

All vehicle control functions, with the exception of the transmission of sonar imaging data, communicate by message passing through that socket. The Strategic Level Prolog is divided into two parts—first the generic mission controller in Table 1, and secondly, the phase level detail in Tables 2–4. For clarity, the higher level rules are highlighted in bold type, the C functions in italics, and any user defined or built-in Prolog predicates are in plain text. The rule set is first compiled and then run in the interpreter by entering the query 'execute_mission.' The example mission outlined in the tables below consists of three phases: *vehicle initialization; submerging to a specified depth while maintaining a heading command; and sweeping the profiling sonar head 360 degrees while still controlling to depth and heading.* This is a deliberately simplified mission for clarity of explanation. Many more complex missions have been run utilizing the principles outlined here.

Referring to the Prolog code in Table 1, the rule head 'execute_mission' will be satisfied, and hence the mission completed if the predicates 'initialize_mission', 'execute_phase', and 'done' all evaluate TRUE. Once 'ood('start_networks', X)' completes, phase 1 is asserted to be the current phase (i.e., set to TRUE), then the entire rule body of 'initialize_mission' is
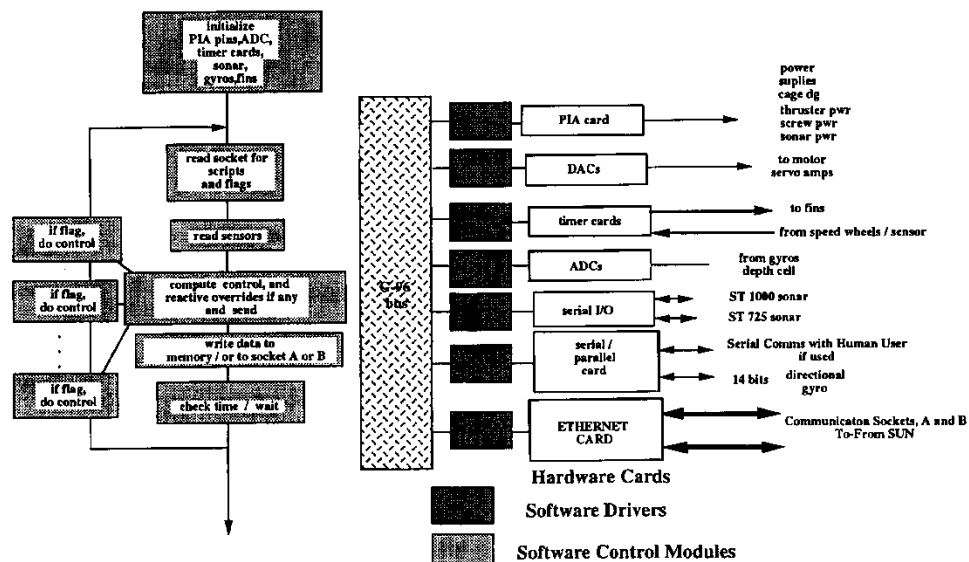

Figure 3. Structure of the execution level software/hardware.

Table 1. Generic mission controller.

done :- current_phase(mission_abort)

done :- current_phase(mission_complete)

execute_mission :- initialize_mission, repeat, execute_phase, done

initialize_mission :- *ood('start_networks', X)*, asserta(current_phase(1)), asserta(complete(0)), asserta(abort(0))

execute_phase :- current_phase(X), execute_phase(X), next_phase(X),!

Table 2. Phase 1 (initialize vehicle systems).

execute_phase(1) :- *exec_init_vehicle(X)*, *exec_start_timer(X)*, repeat, phase_completed(1)

phase_completed(1) :- *ask_init_vehicle_done(X)*, X==1, asserta(complete(1))

phase_completed(1) :- *ask_time_out(X)*, X==1, asserta(abort(1))

next_phase(1) :- complete(1), retract(current_phase(1)), asserta(current_phase(2))

next_phase(1) :- abort(1), retract(current_phase(1)), asserta(current_phase(mission_abort))

Table 3. Phase 2 (submerge to depth and rotate to heading).

execute_phase(2) :- *exec_get_setpoints(X)*,   *exec_submerge(X)*, X==1,   *exec_rotate(X)*, X==1,
                      ***exec_start_timer(X)*,   repeat,   phase_completed(2)**

phase_completed(2) :- *ask_depth_reached(X)*, X==1, *ask_heading_reached(X)*, X==1, asserta(complete(2))

phase_completed(2) :- *ask_time_out(X)*, X==1, *exec_surface(X)*, repeat, *ask_surface_reached(X)*, X==1, asserta(abort(2))

phase_completed(2) :- *ask_sys_problem(X)*, X==1, *exec_surface(X)*, repeat, *ask_surf_reached(X)*, X==1, asserta(abort(2))

next_phase(2) :- complete(2), retract(current_phase(2)), asserta(current_phase(3))

next_phase(2) :- abort(2), retract(current_phase(2)), asserta(current_phase(mission_abort))

Table 4. Phase 3 (sweep sonar).

execute_phase(3) :- *exec_get_setpoints(X)*,   *exec_submerge(X)*, X==1,   *exec_rotate(X)*, X==1,
                      ***exec_set_sonar_mode(X)*, *exec_start_timer(X)*,   repeat,**
                      phase_completed(3)

phase_completed(3) :- *ask_sonar_sweep_complete(X)*, X==1, asserta(complete(3))

phase_completed(3) :- *ask_time_out(X)*, X==1, *exec_surface(X)*, repeat, *ask_surface_reached(X)*, X==1, asserta(abort(3))

phase_completed(3) :- *ask_sys_problem(X)*, X==1, *exec_surface(X)*, repeat, ask_surface_reached(X), X==1, asserta(abort(3))

next_phase(3) :- complete(3), retract(current_phase(3)), asserta(current_phase(mission_complete))

next_phase(3) :- abort(3), retract(current_phase(3)), asserta(current_phase(mission_abort))

evaluated as TRUE. This action enables the control to enter a repeat loop which executes the predicate 'execute_phase' attempting to evaluate each predicate current_phase(X), execute_phase(X), and next_phase(X), as X assumes the values 1 through 3 in sequential order. This particular mission has only three phases, but is expandable to include as many phases as desired.

Each phase includes a 'repeat' predicate so that the rules for phase completion are evaluated repetitively until one of the rules is TRUE. With the exception of vehicle initialization, each phase can terminate in one of three ways:

1. Normal Completion.
   Next Action: (Execute Next Phase)
2. Abort Due to Time Out.
   Next Action: (Surface Immediately)
3. Abort Due to System Problem.
   Next Action: (Surface Immediately)

If phase X completes normally, 'complete(X)' is asserted and X is incremented by one to execute the next phase. Normal completion usually indicates that

Table 1. Generic mission controller.

done :- current_phase(mission_abort)

done :- current_phase(mission_complete)

execute_mission :- initialize_mission, repeat, execute_phase, done

initialize_mission :- ood('start_networks', X), asserta(current_phase(1)), asserta(complete(0)), asserta(abort(0))

execute_phase :- current_phase(X), execute_phase(X), next_phase(X),!

Table 2. Phase 1 (initialize vehicle systems).

execute_phase(1) :- exec_init_vehicle(X), exec_start_timer(X), repeat, phase_completed(1)

phase_completed(1) :- ask_init_vehicle_done(X), X==1, asserta(complete(1))

phase_completed(1) :- ask_time_out(X), X==1, asserta(abort(1))

next_phase(1) :- complete(1), retract(current_phase(1)), asserta(current_phase(2))

next_phase(1) :- abort(1), retract(current_phase(1)), asserta(current_phase(mission_abort))

Table 3. Phase 2 (submerge to depth and rotate to heading).

execute_phase(2) :- exec_get_setpoints(X),   exec_submerge(X), X==1,   exec_rotate(X), X==1,
exec_start_timer(X),   repeat,   phase_completed(2)

phase_completed(2) :- ask_depth_reached(X), X==1, ask_heading_reached(X), X==1, asserta(complete(2))

phase_completed(2) :- ask_time_out(X), X==1, exec_surface(X), repeat, ask_surface_reached(X), X==1, asserta(abort(2))

phase_completed(2) :- ask_sys_problem(X), X==1, exec_surface(X), repeat, ask_surf_reached(X), X==1, asserta(abort(2))

next_phase(2) :- complete(2), retract(current_phase(2)), asserta(current_phase(3))

next_phase(2) :- abort(2), retract(current_phase(2)), asserta(current_phase(mission_abort))

Table 4. Phase 3 (sweep sonar).

execute_phase(3) :- exec_get_setpoints(X),   exec_submerge(X), X==1,   exec_rotate(X), X==1,
exec_set_sonar_mode(X), exec_start_timer(X),   repeat,
phase_completed(3)

phase_completed(3) :- ask_sonar_sweep_complete(X), X==1, asserta(complete(3))

phase_completed(3) :- ask_time_out(X), X==1, exec_surface(X), repeat, ask_surface_reached(X), X==1, asserta(abort(3))

phase_completed(3) :- ask_sys_problem(X), X==1, exec_surface(X), repeat, ask_surface_reached(X), X==1, asserta(abort(3))

next_phase(3) :- complete(3), retract(current_phase(3)), asserta(current_phase(mission_complete))

next_phase(3) :- abort(3), retract(current_phase(3)), asserta(current_phase(mission_abort))

---

evaluated as TRUE. This action enables the control to enter a repeat loop which executes the predicate 'execute_phase' attempting to evaluate each predicate current_phase(X), execute_phase(X), and next_phase(X), as X assumes the values 1 through 3 in sequential order. This particular mission has only three phases, but is expandable to include as many phases as desired.

Each phase includes a 'repeat' predicate so that the rules for phase completion are evaluated repetitively until one of the rules is TRUE. With the exception of vehicle initialization, each phase can terminate in one

of three ways:

1. Normal Completion.
   Next Action: (Execute Next Phase)
2. Abort Due to Time Out.
   Next Action: (Surface Immediately)
3. Abort Due to System Problem.
   Next Action: (Surface Immediately)

If phase X completes normally, 'complete(X)' is asserted and X is incremented by one to execute the next phase. Normal completion usually indicates that

a commanded set point or task has been accomplished and the vehicle is ready to start the next phase. In the case of phase 2, this means that the commanded depth and heading has been achieved. If a time out or a system problem occurs, the vehicle is commanded to surface immediately and a mission abort for phase X is asserted after the surface is reached. A time out indicates that a set point or task is taking too much time to complete and with our current version of error recovery the mission phase is aborted and also the entire mission. System problems can cover a variety malfunctions, sensor failures, thruster failures, or any type of hardware problem, which are assumed to be catastrophic requiring an entire mission abort.

After each phase executes, the predicate 'done' is evaluated. If the next phase is commanded, 'done' fails and the cycle continues, if however a mission abort is asserted or the mission completes, 'done' is satisfied and 'execute_mission' evaluates TRUE and the entire mission is finished.

Tasks that are required to be performed in successive phases are recommanded as shown by the calls "..*exec_submerge(X)*, X==1, *exec_rotate(X)*, X==1, ..," which appear in Phase 2 and 3 (Table 3 and 4). In this way new set points can be entered for each phase. Generally, control mode commands are left active until changed although 'kill' rules can be used to stop actions such as filters etc.

Relations between Prolog and Petri Nets

So far the mission controller has been represented with Prolog. While this is the actual language that is used to drive the Strategic Level in real time, there also exists a method to graphically model discrete mission events. These models are referred to as Petri nets (Murata, 1989), and can sometimes give a more clear and intuitive representation for a Strategic Level mission controller. It should be pointed out that using a Petri net graph is not intended to replace the Prolog code, but rather provides a different representation of the mission controller. The following sections show how Prolog code may be written given a Petri net graph, followed by a Petri net analog of the Prolog mission code outlined before. It should be noted that our use of Prolog is greatly simplified and so are the Petri net representations, being limited to single token places corresponding to the TRUE/FALSE evaluations of the predicate rules.

Three example Petri net graphs are shown in Figs. 4–6. The circles denoted by P_1, P_2, . . . , P_n are the states
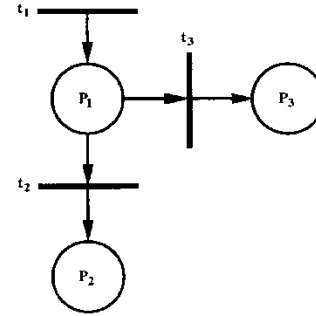

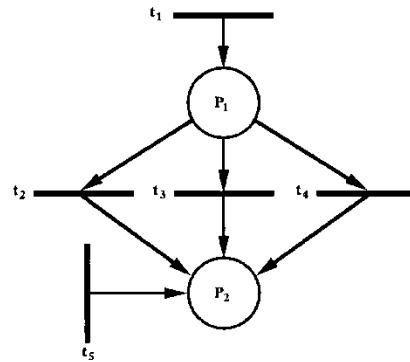*Figure 4.* Petri net graph for Example 1.
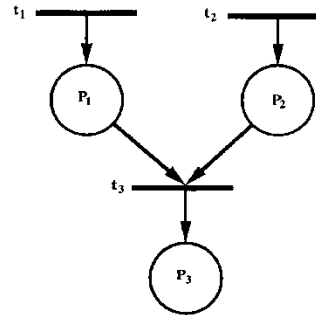

Figure 5. Petri net graph for Example 2.

Figure 6. Petri net graph for Example 3.

or places of the Petri net graph and the bars labeled t_1, t_2, . . . , t_n are the transitions, where a token enters a place whenever a transition is fired. The basic technique for writing Prolog code from an existing Petri net is manually done and not optimized, but to

Table 5. Prolog for Example 1.

mission_complete :- repeat, done

done :- p2

done :- p3

t1

p1 :- t1

p2 :- p1, ask('t2(X)', X), X==1

p3 :- p1, ask('t3(X)', X), X==1

ask(Q,A) :- write(Q), write('?'), nl, read(A), nl

Table 6. Prolog for Example 2.

mission_complete :- repeat, done

done :- p2

t1

p1 :- t1

p2 :- p1, ask('t2(X)', X), X==1

p2 :- p1, ask('t3(X)', X), X==1

p2 :- p1, ask('t4(X)', X), X==1

p2 :- ask('t5(X)',X), X==1

ask(Q,A) :- write(Q),write('?'), nl, read(A),nl

Table 7. Prolog for Example 3.

mission_complete :- repeat, done

done :- p3

p1 :- ask('t1(X)', X), X==1

p2 :- ask('t2(X)', X), X==1

p3 :- p1, p2, ask('t3(X)', X), X==1

ask(Q,A) :- write(Q),write('?'), nl, read(A),nl

start at the terminal state(s) and work back towards the initial state of the graph (following the backward chaining nature of Prolog). While doing this we define 'place' and 'transition' predicates, where 'place' predicates evaluate TRUE if a token resides there, and a 'transition' predicate is TRUE if it is enabled and the transition has fired.

Using this approach, Prolog code has been generated for three example Petri net graphs (Tables 5 through 7). Each Prolog example is driven by executing 'mission_complete' which is the rule to be satisfied for completion with the predicate 'done' repeatedly queried until TRUE. The 'place' and 'transition' predicates are denoted by $p(n)$ and $t(n)$ respectively. The 'ask' predicate shown in these examples allows the user to interactively activate the firing of a transition by typing a 1 for TRUE or 0 for False.

The Petri net of Example 1 shows two terminal states, P_2 and P_3, which implies that for completion, two instances of the Prolog 'done' predicate must be defined, (done :- p2. or done :- p3.). Starting with the terminal states and working back, we must determine what conditions must be satisfied to reach these states. The precondition for a transition to either P_2 or P_3 requires a token in place P_1. This is assured since the rule for transition t_1 is declared to be TRUE. At this point both transitions t_2 and t_3 are 'enabled' and either may fire to move the token to terminal states P_2 or P_3.

Example 2 is an OR structure with a single terminal state P_2 (done :- p2.) which may be reached one of four ways. The first three through transitions t_2, t_3, or t_4, and a fourth, directly through t_5. This requires the Prolog to have four instances of the rule P_2, which reflects the OR nature of the Petri net.

Example 3 is an AND structure with a single terminal place, P_3 (done :- p3.), where both places P_1 and P_2 must be occupied to enable transition t_3. This is described in Prolog by the AND (p1, p2) in the rule body of p3.

In our mission, completing a phase normally, a phase time out, or having a system problem are all examples of a discrete event. In a Petri net graph, these are the transitions, the places represent the execution of mission commands such as 'submerge', 'rotate', etc. A Prolog 'repeat' loop is also considered to be a place with it's transition predicates continually evaluated. Fig. 7 shows the Petri net graph for the mission, which starts with a transition at t_1 (equivalent to querying the rule 'execute_mission'). The places P_1, P_2 and P_3 represent the Prolog phases 1, 2 and 3 and are expanded in detail in Figs. 8, 9 and 10 respectively. The transitions t_2, and t_5 through t_{10} denoted with a thick line are only evaluated if enabled, and fired when the associated predicate becomes TRUE, (X==1). A transition drawn with a thin line fires as soon as it is enabled. In Petri net notation, we are using a 'timed' graph since there are definite time delays between the enabling and subsequent firing of some—but not all—transitions. Referring to the expanded Petri net graph of phase 2 in Fig. 9, the transition t_{21} will fire when the rule 'execute_phase(2)' is executed. Once this has occured, the predicates represented by P_{22}, P_{23}, and P_{24} are all executed and upon completion, the transition t_{22} fires immediately and the place P_{25} becomes active. At this point the transitions t_6, t_7, t_{23}, and t_{24} are enabled and the predicates associated with them are evaluated repetitively until one of them is TRUE, at which time the respective transition
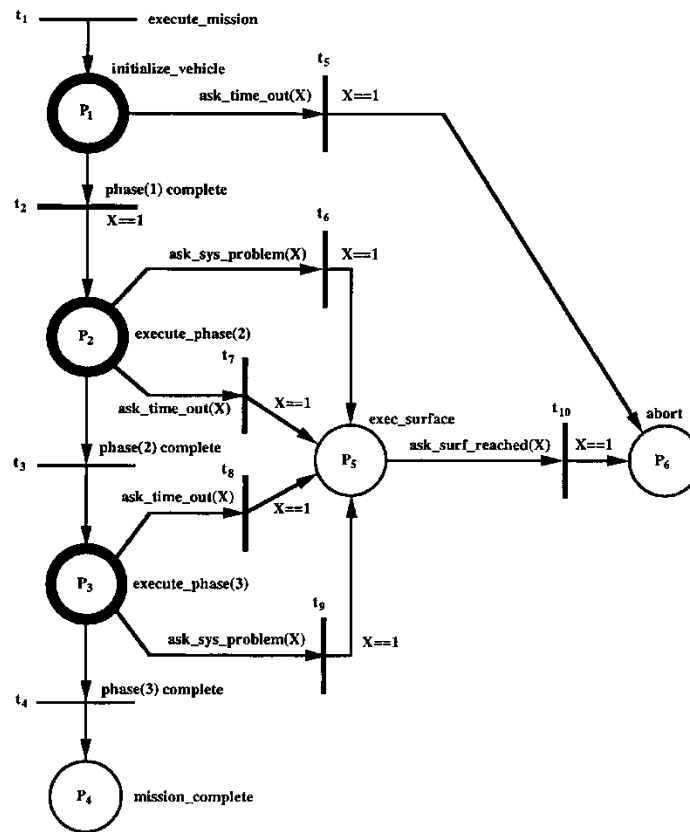
Figure 7. Petri net graph for generic mission.

will fire. If a time out or system problem does not oc-
cur and the desired depth is reached, t_{23} will fire, and
the predicate 'ask_heading_reached' must also be eval-
uated repetitively. Since the Prolog repeat continues
to evaluate transitions, t_{23} must return a token to P_{25}
causing t_6, t_7, t_{23}, and t_{24} to remain enabled and eval-
uated. When the heading is reached, t_{24} is fired, t_{25}
is enabled, and fires immediately completing phase 2
normally. A similar structure of phase completion and
error recovery is used in phase 3 as well, which can
serve as a template for most any mission phase.

If a time out or system problem occurs in either
phase, it is clearly shown that one of the transitions
t_6 through t_9 will fire and the 'exec_surface' predicate
(place P_5) will be executed. In this state the predi-
cate 'ask_surf_reached' is continually queried until the

surface is reached at which time the mission is aborted
(P_6). If all goes well and the objectives of phases 2 and
3 are met, the place P_4 (mission_complete) is reached
and the mission terminates normally.

Tactical Level Software

The Tactical Level of the control system contains all
the C functions that are compiled as predicates in the
Strategic Level rules, and performs the computations
upon which the vehicle commands and transitions are
based. Additionally, a second Sparc process called
the 'Sonar Manager' is opened which runs asyn-
chronously in the Sparc and with equal priority to the
'Mission_Control'. This process is linked through a
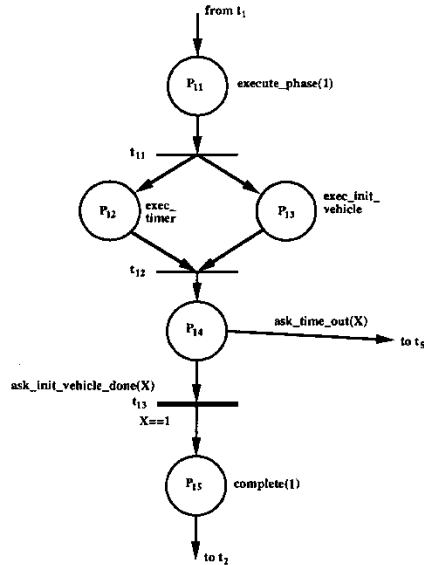separate socket ('B' in Fig. 2) to the Gespac for the

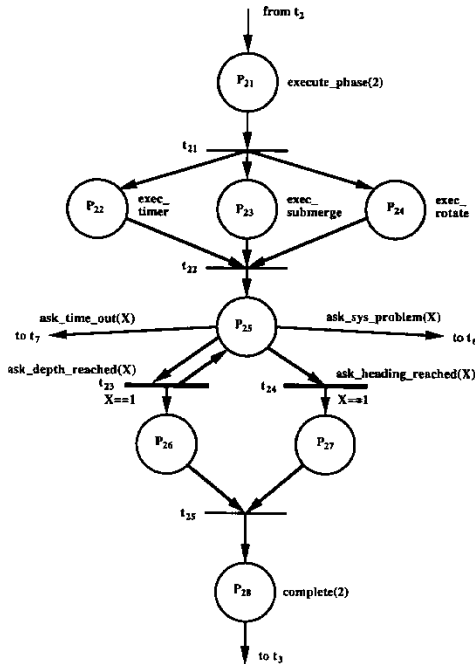Figure 8. Petri net graph for phase 1.

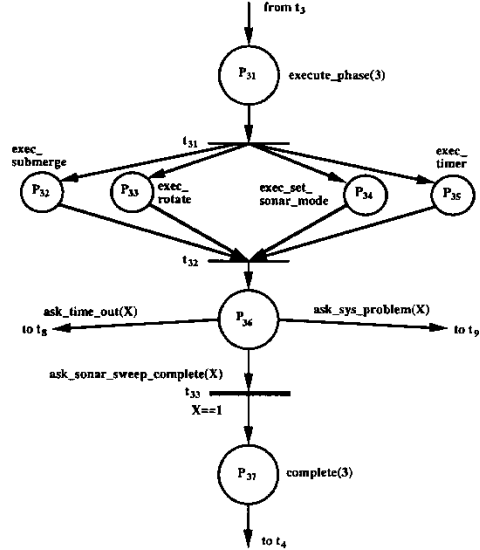Figure 9. Petri net graph for phase 2.

Figure 10. Petri net graph for phase 3.

purpose of the reception and handling of sonar imaging data. The **'Sonar Manager'** captures data that is sent out from the Execution Level as soon as it has been acquired, and then processes and passes the data to be displayed on the IRIS Graphics workstation for visualization purposes.

The introduction of the additional process called **'Sonar Manager'** and it's separation from the **'Mission_Control'** Tactical Level functions has been found to be important and a necessary first step toward a more general Concurrent Tactical Level that was foreseen by the earlier RBM architecture (Byrnes et al., 1993) and explained recently by Kwak and Thornton (1994). The need for concurrency of multiple processes lies fundamentally with the fact that sonar data is obtained *asynchronously* with bounded but unknown latency and the servo control functions cannot wait for the sonar port data to arrive. While it is perfectly normal to send control set point commands *asynchronously* to stable control loops, waiting for sonar returns could hold up the servicing of the inner servo loop commands to actuators. Thus in our solution to this problem, we have defined the additional **'Sonar Manager'** process to always read the socket onto which sonar data is written so that it is immediately free for another sonar write without delay and the servo loop is made independent of direct involvement with the sonar. As an unpleasant

alternative, we have found that without the **'Sonar Manager'**, all the Tactical Level functions would have to be modified to include a check to read sonar data. This would have been a cumbersome addition of much unnecessary code writing.

Transition Criteria

Most control phase transitions of the Phoenix are event based, meaning that a certain set of criteria must be met in order for a transition to occur. A common example of this is when a position set point is sent to the vehicle controllers and reached. A method of determining whether the vehicle has indeed reached this point must be programmed into the control logic. Measuring the position error alone and declaring the maneuver complete when this error is small is not sufficient. This is because the vehicle could be overshooting the commanded position and simply passing through the set point. Therefore, not only must the position error be small but the rate error must also be small. This dual criteria can be expressed mathematically as a positive definite, linear combination of the position error e and the position rate error $\dot{e}$. We use,

$$\sigma_k = w_e|e_k| + w_{\dot{e}}|\dot{e}_k| \tag{1}$$

where w_e and $w_{\dot{e}}$ are positive weights for the position and rate errors respectively. This equation allows a minimum value of σ, denoted σ_0, to be specified defining a threshold for the combination of errors which can be set relatively large when precision control is not required or low for extremely precise positioning. Once σ drops below σ_0, the maneuver is declared complete and a transition to the next control phase may occur.

When noisy sensors are used, the noise prevents σ from settling enough to determine an accurate measurement for the transition, and the use of Eq. (1) alone has shown to be unsatisfactory. The signal can be smoothed by filtering σ through a first order digital filter of the form

$$\sigma_{f(k+1)} = e^{-T/\tau}\sigma_{f(k)} + (1 - e^{-T/\tau})\sigma_k \tag{2}$$

where σ_f is the filtered form of σ, τ is the time constant of the filter, and T is the sampling time. The condition for transition can be shown diagrammatically in Fig. 11, which indicates that the signal for transition, s, is 1 (TRUE) for $\sigma_f < \sigma_{f0}$ or 0 (FALSE) for $\sigma_f > \sigma_{f0}$. As an example, the function 'ask_depth_reached(X)',
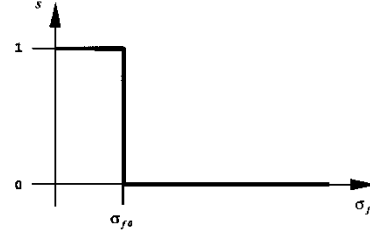
Figure 11. Transition signal condition.

performs the calculations above and returns s. Other dynamic error and time based signals are computed similarly.

Execution Level Software

The structure of the Execution Level software is illustrated by Fig. 3 which indicates that it is composed of software at the hardware interface (software drivers) as well as software for vehicle control. After initialization of power systems and sonars, and the basic driver settings, the PIA card pins that control the on/off feature of power supplies, thruster power, screw power, and sonar power, a simple timing loop is entered and reentered at a fixed update rate (in our case 0.1 sec.) during which the following takes place,

1. read the socket 'A' for behavior based mode command flags and control set points,
2. read all sensors,
3. select appropriate 'C' code control functions for computing and sending control values to actuators, using an if-else structure for distinguishing the commands,
4. write selected data to memory or sockets 'A' or 'B' as appropriate, and
5. as appropriate, send signals to the Sonars to ping and rotate,
6. check time for any time based events and wait for the next timing interrupt to maintain integrity of the digital control loop.

Specific control laws as built into callable modules of code are easily selected according to the communication flags, provided that they exist in the first place.

An example of the 3 levels of control interactions can be seen from the following code fragments.

STRATEGIC LEVEL (PROLOG)

.

...exec_submerge(X), ...

.

...repeat, ask_depth_reached(X),...

.

.

TACTICAL LEVEL (C)

```
int exec_submerge()
{
    sprintf(command_sent,"%s %f %f","SUBMERGE", z_setpt[current_setpt_index],
        theta_setpt[current_setpt_index]); /* Command vehicle to Submerge to z_setpt */
    write_to_execution(command_sent);
    return(TRUE);
}

int ask_depth_reached()
{
    sprintf(command_sent,"%s","GET_DEPTH_INFO");
    write_to_execution(command_sent);   /* Request Depth Information from Execution Level */
    read_from_execution(&command_read[0]);    /* Read Reply from Execution Level, blocking socket */
    sscanf(command_read,"%F %F",&z_est, &sigma_zf);

    if( sigma_zf < sigma_zf_min[current_setpt_index] )
    {
        return(TRUE); /* Within Minimum Error */
    }
    else
    {
        return(FALSE); /* Outside Minimum Error */
    }
}
```

EXECUTION LEVEL (C)

```
while (shutdown_signal_received == FALSE)
{
    /* Read Command (If Any) From Tactical Level */
    read_status = read_from_tactical(&command_read[0]);    /* non-blocking socket read */
    if(read_status > 0)
    {
        sscanf(command_read,"%s",&command[0]); /* Extract the Command Only! */
        /* Switch to the Appropriate Command Parser */
        if(!strcmp(command,"SUBMERGE"))
```

107

```
        {
                sscanf(command_read,"%s %F %F", &command[0],&z_com,&theta_com);
                DEPTH_AND_PITCH_CONTROL = TRUE;
        }
        else if(!strcmp(command, "GET_DEPTH_INFO"))
        {
                sprintf(command_sent,"%f %f", z_est,sigma_zf);
                write_to_sun(command_sent);
        }
    }
    /* Control Block */

    if(DEPTH_AND_PITCH_CONTROL)
    {
        submerge_and_pitch_control(z_com,theta _com);
    }
}
```

This shows how the Strategic Level communicates with the Tactical Level which in turn sends command strings to the execution level for submerging control. When the Prolog predicate 'exec_submerge(X)' is executed, the 'C' function in the Tactical Level is called which writes the command SUBMEGE along with the depth and pitch angle set point for the particular phase to the Execution Level. This function then completes and having sent the command to the execution level returns a state of TRUE. At this time the Execution Level extracts which command has been sent and program control is switched to the appropriate command parser block. Since SUBMERGE was sent, the command parser expects two set points, depth and pitch angle. Once this command has been received, a flag DEPTH_AND_PITCH_CONTROL is set TRUE which activates this control function and will remain in effect until commanded otherwise.

Socket Communications (Tactical/Execution Level)

Careful attention must be paid to both sides of a communications socket when dealing with synchronous and asynchronous processes. Reading from a 'blocking' socket causes execution to pause until data is received. In contrast to that, a 'non-blocking' socket allows execution to proceed if no data is waiting to be read. For synchronous real time execution of dynamic processes attempting to make a read every time step, a 'non-blocking' socket is a requirement. Since the Tactical Level sends commands and receives data

asynchronously, while the Execution Level must run synchronously at 10 Hz., the UNIX side of socket A is configured to be 'blocking', while the OS-9 side is 'non-blocking'. By contrast, eight different types of socket communications are used by the Esterel language mentioned previously (Simon et al., 1993).

Execution of the predicate 'ask_depth_reached(X)' sends a request to the Execution Level for depth information (GET_DEPTH_INFO). The command is parsed in exactly the same way as before except that the Tactical Level function waits ('blocking' socket) for the Execution Level to return the values of depth and filtered depth error. A comparison is then made between the current filtered error, σ_{zf}, and the prespecified minimum, σ_{zf0}. and the function returns TRUE or FALSE as appropriate.

Human Supervision

Human supervisory control has not been built into the control system to date. This does not mean that it is impossible to do. In fact, user inquiry for the state of the vehicle can easily be incorporated into a tactical level function that reads an acoustic modem and waits for messages to be received. The Strategic Level predicates can include a predicate that asks if a user message has been received. The Tactical Level message can be parsed into commands that could call any of the vehicle primatives directly—or specifically—request data to be changed. While the architecture supports supervisory control, that is not the main focus of our work to date.

4. Results from Experimental Mission

The mission described in this paper was performed in the NPS hover tank which measures 6.0 by 6.0 meters square and 1.8 meters deep. During execution all pertinent data was collected, including depth, heading, thruster motor speed, etc. During phases where the sonar is active, the range and heading angle of the sonar head was recorded. A log file of mission status messages, a time history of the depth response of all three phases, and a plot of the profiling sonar image of the tank was obtained.

While the mission executes, the process running the Strategic Level displays status messages to the screen, while others are written by the Tactical Level C functions. Stored in a log file for each mission, the following was obtained with messages from the Strategic Level in upper case and in lower case for the Tactical Level.

?- execute_mission.

INITIALIZE MISSION!
START NETWORK!
READ MISSION FILE!
2
60.0 0.0 0.0 2.5 0.0 0.0 0.0 0.0 0.0 0.1 0.0 0.0 0.1
    0.0 0 0.0 0.0
60.0 0.0 0.0 2.5 0.0 0.0 0.0 0.0 0.0 0.1 0.0 0.0 0.1
    60.0 1 0.0 0.0 : mission file
Mission File opened successfully.

START PHASE 1!
INITIALIZE VEHICLE!
INITIALIZE BOARDS!
TURN ON PROP POWER!
TURN ON SONAR POWER!
UNCAGE DIRECTIONAL GYROSCOPE!
DIRECTIONAL GYROSCOPE UNCAGED.
ZEROING SENSORS.
INITIALIZE ST1000 SONAR!
INITIALIZATION DONE.
PHASE 1 COMPLETE.

START PHASE 2!
current_setpt_index = 0
SUBMERGE!
z_setpt = 2.5 theta_setpt = 0.0

ROTATE!
psi_setpt = 0.0

START_TIMER!

DEPTH TRANSITION.
Depth @ transition = 2.401626
z_dot @ switch = 0.004895
sigma_zf @ transition = 0.093

HEADING TRANSITION.
Heading @ switch = −0.033422
r @ switch = 0.003719
sigma_psif @ transition = 0.084
PHASE 2 COMPLETE.

START PHASE 3!
START SWEEP TIMER!
SET SONAR MODE!
START TIMER!
SONAR SWEEP COMPLETE.
PHASE 3 COMPLETE.

DIS-CONNECT NETWORKS!
MISSION COMPLETE.

yes
| ?-

Note: The log file uses units of feet, feet/sec, radians, and radians/sec.

The commanded depth was 0.762 m (2.5 feet) with a filtered error threshold 0.03 m (0.1 feet). The set points for both pitch and heading angle were 0.0 radians, and the sonar was set to continuously sweep clockwise (Mode +1) for 60.0 seconds in phase 3. After the network connections to the various processes were established, the mission file was read by the Tactical Level. Although this was a three phase mission, only two rows of set points were required. Vehicle initialization does not require set point data.

The first column of the mission data file is the time out for a phase (seconds), the next six columns are the set points for longitudinal, lateral, depth, roll, pitch, and heading positions. The second set of six columns are their respective filtered error thresholds, σ_{f0}, and the last four columns contain the duration of the sonar sweep, the sweep mode, scan direction, and sweep width. The log file showed the status of the various transitions and numerical values for certain variables of interest. Upon completion of phase 3, the network connections are terminated and the mission completes.
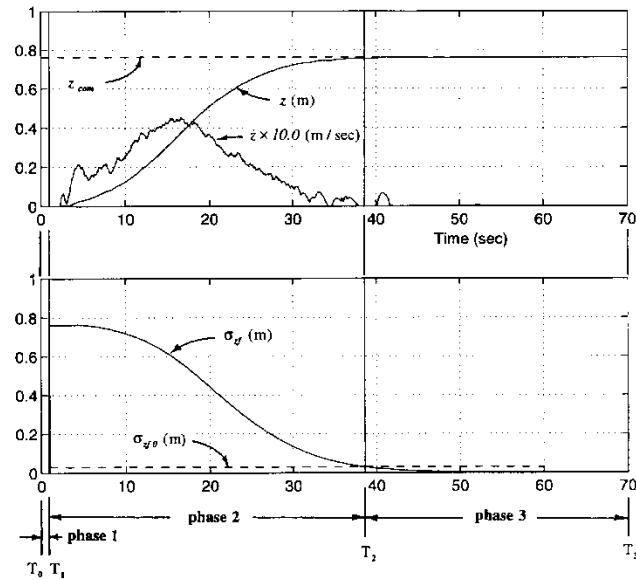
109

Figure 12. Vehicle depth response for all three phases.

Figure 12 shows the time history of the depth and depth rate response. The lower trace shows the behavior of the filtered depth error, σ_{zf}, and the threshold for the filtered error, σ_{zf0}. The time axis includes a short time for initialization, and in phase 2 it can be seen that σ_{zf} starts to reduce as the vehicle begins to submerge at T_1. The transition to phase 3 is triggered as σ_{zf} reaches σ_{zf0} (T_2), when the sonar is activated and an image of the test tank walls and a cylindrical object is recorded as shown by Fig. 13. While this phase is active, the depth controller continues to operate and reduces the error beyond the threshold of 0.03 m to nearly zero. Once the sonar sweep time is over, the mission terminates at time T_3.

It is not an easy task to evaluate a given control system architecture. The theoretical design for stability and robustness leads to selection of parameters that are used in the control functions of the Execution Level. We are going beyond that now and are interested in the organization of control software. Some software controllers will be successful for fixed purpose tasks, but here, we have a multipurpose flexible control requirement and, because we are talking about control software, we are led to ask the following questions,
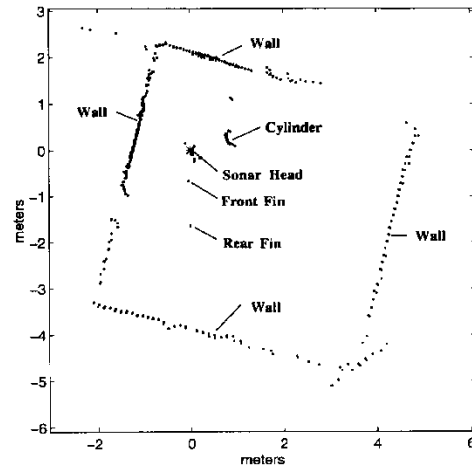

Figure 13. Sonar image of test tank.

1. Does the controller permit easy evaluation of response and change to control parameters to 'tune' the low level servos?

2. Can this be done while testing is ongoing in real time?

3. Can new sensors be added to the vehicle with little change to the control software?

4. What levels of code and how many functions have to be changed for this new sensor to be added?

5. How many rules (code statements) must be changed if the mission is altered to eliminate, or add, a new phase?

6. How is the control code modified to test just the performance of a particular existing sensor or actuator set?

7. How easy is it to change the data record for a different set of sensors?

8. How easy is it to change the conditions that define the transition signals?

These questions are currently being evaluated. In particular, this paper deals with a mission that has three phases; the initialization, the submergence, and the sonar mapping phases. Although the mission here was simplified so that the details of the code and results could be more clearly presented, other more complex missions have been performed successfully.

Conclusion

The conclusion of our work to date has indicated that complex behavior can be readily coordinated through Strategic Level rules, that are easily modified. These act as state transitioning mechanisms and the communication through Tactical Level software to the Execution Level controllers is a simple but convenient way of commanding stable responses of the vehicle. The design of well behaved control laws and functions at the Execution Level is essential as a primary part of the design and is affected through careful attention to the digital control loop design. Reactivity, failure recovery, and even human interfacing within the controller can take place at any level.

Acknowledgment

References

Albus, J. 1988. System description and design architecture for multiple autonomous undersea vehicles. National Institute of Standards and Technology, Technical Note 1251.

Albus, J. and Quintero, R. 1990. Towards a reference model architecture for real-time intelligent control systems (ARTICS), In Robotics and Manufacturing, New York, ASME. vol. 3.

Antsaklis, P.J. and Passino, K.M. 1993. An Introduction to Intelligent and Autonomous Control, Kluwer Academic Publishers: ISBN #0-7923-9267-1.

Bellingham, J.G. and Consi, T.R. 1991. State configured layered control. Procs of Mobile Robots for Subsea Environments, IARP, Monterey, CA, pp 75–79.

Bellingham, J.G. and Goudey, C.A. et al. 1994. A second generation survey AUV. Procs. IEEE Symp. on Auton. Underwater Vehicle Tech., pp. 148–156.

Brooks, R. 1986. A robust layered control system for a mobile robot. IEEE J. Rob. and Autom., RA-2(1).

Byrnes, R.B., MacPherson, D.L., Kwak, S.H., McGhee, R.B., and Nelson, M.L. 1992. An experimental comparison of hierarchical and subsumptiom software architecture for control of an autonomous underwater vehicle. Proc. IEEE Symposium on AUV Technology, Washington, DC, pp. 135–141.

Byrnes, R., Kwak, S.H., McGhee, R.B., Healey, A.J., and Nelson, M. 1993. Rational behavior model: An implemented tri-level multilingual software architecture for control of autonomous vehicles. Proceedings of 8th International Symposium on Unmanned Untethered Submersible Technology, Durham, New Hampshire, pp. 160–179.

Cassandras, C.G. 1993. Discrete event systems, modeling and performance analysis. Aksen Associates, ISBN #0-256-11212-6.

Curtin, T.B., Bellingham, J.G., Catopovic, J., and Webb, D. 1993. Autonomous oceanographic sampling networks. Oceanography, 6(3):86–94.

Freidland, B. 1986. Control System Design: Introduction to State Space Methods, McGraw Hill: ISBN #0-07-022441-2.

Hall, D. and Adams, M. 1992. Autonomous Vehicle Software Taxonomy. Procs. IEEE Symp. on AUV Tech., pp. 49–64.

Healey, A.J. and Marco, D.B.1992a. Experimental verification of mission planning by autonomous mission execution and data visualization using the NPS AUV II. Proceedings of IEEE Oceanic Engineering Society, Symposium on Autonomous Underwater Vehicles, AUV-92, Washington, DC.

Healey, A.J. and Marco, D.B. 1992b. Slow speed flight control of autonomous underwater vehicles: Experimental results with NPS AUV II. Proceedings of the 2nd International Offshore and Polar Engineering Conference, San Francisco.

Healey, A.J., et al. 1994. Tactical / Execution Level coordination for hover control of the NPS AUV II using onboard sonar servoing. Proceedings of the IEEE Symposium on Autonomous Underwater Vehicle Technology, Cambridge, Mass. pp. 129–138.

Healey, A.J., McGhee, R.B., Christi, R., Papoulias, F.A., Kwak, S.H., Kanayama, Y., and Lee, Y. 1991. Mission planning, execution and data analysis for the NPS AUV II autonomous underwater Vehicle. Procs. of 1st IARP Workshop on Mobile Robots for Subsea Environments, MBARI, Pacific Grove, CA, pp. 177–186.

Kwak, S.H. and Thornton, F.P.B. 1994. A concurrent object-oriented implementation for the tactical level of the rational behavior model software architecture for UUV control. Proceedings of the IEEE Symposium on Autonomous Underwater Vehicle Technology, Cambridge, Mass., pp. 54–60.

Moore, J., (Ed). 1994. Commercialization of Autonomous Underwater Vehicles. Report No. MITSG 93-32, MIT Sea Grant, Cambridge, Massachusetts.

Murata, T. 1989. Petri nets: Properties, analysis, and applications.

111

Proceedings of IEEE, Vol. 77, pp. 541–580.

Newman, J.B. and Stakes, D. 1994. Tiburon: development of an ROV for ocean science research. *Proc. OCEANS'94*, Brest, France, vol. II, pp. 483–488.

Saridis, G.N. 1989. Analytical formulation of the principle of increasing precision with decreasing intelligence for intelligent machines. *Automatica*, 25:461–467.

Silva, V., Oliveira, P., Silvestre, and C., Pascoal, A. 1994. Mission coordination synthesis and execution using the CORAL language. Report 1994/08/04 Instituto de Systemas e Robotica, Instituto Superior Technico, Lisbon, Portugal.

Simon, D., Espiau, B., Castillo, E., and Kapellos, K.1993. Computer aided design of a generic robot controller handling reactivity and real time control issues. *IEEE Transactions on Control Systems Technology*, 1(4):213–229.

Smith, S.M. and Dunn, S. 1994. The ocean voyager II: An AUV designed for coastal oceanography. *Procs. IEEE Symp. on Auton. Underwater Vehicle Tech.*, pp. 139–148.

Sousa, J.B., Pereira, F.L., and Silva, E.P. 1994. A dynamically configurable architecture for the control of an AUV. *Procs. OCEANS '94*, Brest, France, pp. 131–136.

Yuh, ed. 1994. Future directions in underwater robotics. NSF Workshop, Maui.
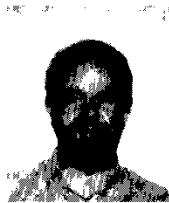
David Bryan Marco received the B.S. degree in Mechanical Engineering from Oklahoma State University in 1983, and the M.S. degree in Mechanical Engineering from the University of Texas at Austin in 1987. He is currently completing the Ph.D. degree in Mechanical Engineering at the US Naval Postgraduate School, Monterey, CA. His areas of interest include control systems, robotics, and underwater vehicle control using multiple, networked, cooperating computers. Present work involves using high frequency sonar for local navigation and precision control of Autonomous Underwater Vehicles. He is a member of the ASME and NSPE.

Anthony J. Healey was graduated from London and Sheffield Universities with the degrees B.Sc.(Eng.) and Ph.D. in Mechanical Engineering in 1961 and 1966 respectively. He emigrated to the US in 1966 and has taught at The Pennsylvania State University, MIT, The University of Texas at Austin, and the Naval Postgraduate School. He was promoted to Full Professor of Mechanical Engineering in 1974 at the University of Texas at Austin, and in 1981, he joined Brown and Root Inc. as manager of the Pipeline and Sub-Sea Technology Re-

search Group. In 1986, he returned to academe and was appointed as Professor of Mechanical Engineering at the US Naval Postgraduate School, serving until 1992 as Department Chairman.

His areas of technical specialty include Control Systems, Marine Vehicle System Dynamics, Robotics, and Vibration. Since 1987, he has led an Interdisciplinary Project in Advanced Controls for Autonomous Underwater Vehicles that encompass the integration of Control Logic for Mission Management including automated error recovery, the use of high frequency sonar in local navigation of AUV's, and the precision control of vehicle motion. He is the Director of the NPS Autonomous Underwater Vehicles Laboratory having built and operated the NPS Phoenix autonomous vehicle as a testbed for experiment evaluation of control concepts. The work of the NPS team has appeared in numerous places in the AUV literature.

He is a member of IEEE and the IEEE Oceanic Engineering Society, a Fellow of the ASME, having held positions within ASME as Chairman of the Dynamic Systems and Control Division, Member of the Systems and Design Technical Board, and is an ASME National Nominating Committee member. He has served the IEEE OES as contributor to the AUV '90, '92, '94 Symposia and was the Technical Program Chairman for the AUV '94 Symposium held in Cambridge, Mass. July 1994, and has recently been a Guest Editor for the *IEEE Journal of Oceanic Engineering*.

Robert B. McGhee was born in Detroit, Michigan in 1929. He received the B.S. degree in Engineering Physics from the University of Michigan in 1952, and the M.S. and Ph.D. degrees in Electrical Engineering from the University of Southern California in 1957 and 1963 respectively.

From 1952 until 1955, he served on active duty as a guided missile maintenance officer with the U.S. Army Ordnance Corps. From 1955 until 1963, he was a member of the technical staff with Hughes Aircraft Company, Culver City, CA, where he worked on guided missile simulation and control problems. In 1963, he joined the Electrical Engineering Department at the University of Southern California as an Assistant Professor. He was promoted to Associate Professor in 1967. In 1968, he was appointed Professor of Electrical Engineering and Director of the Digital Systems Laboratory at Ohio State University, in Columbus, OH. In 1986, he joined the Naval Postgraduate School in Monterey, CA, as Professor of Computer Science. He served as Chairman of the Computer Science Department from 1988 until 1992.

Dr. McGhee is a Fellow of the Institute of Electrical and Electronic Engineers. He currently teaches in the areas of artificial intelligence, robotics, computer graphics, and feedback control theory. His research interests are centered around computer simulaton and control of unmanned vehicles, especially for subsea applications.